Performance Tools and Analysis

Scott Parker

Getting Started Workshop: Oct 3 & 4, 2011

Argonne Leadership Computing Facility

Performance Tools

- Performance tools collect information during the execution of a program to enable the performance of an application to be understood and possibly improved
- Different tools collect different information and collect it in different ways
- It is up to the user to determine what information to collect, what tool to use, how to interpret the collected information, and how to change code to improve the performance

Types Performance Information

- Types of information collected by performance tools:
  - Time in routines and code sections
  - Call counts for routines
  - Call graph information with timing attribution
  - Information about arguments passed to routines:
    - MPI – message size
    - IO – bytes written or read
  - Hardware performance counter data:
    - FLOPS
    - Instruction counts (total, integer, floating point, load/store, branch, …)
    - Cache hits/misses
    - Memory, bytes loaded and stored
    - Pipeline stall cycle
  - Memory usage

Sources of Performance Information

- Instrumentation:
  - Insert calls to performance collection routines into the code to be analyzed
  - Allows a wide variety of data to be collected
  - Can only be used on routines that can be edited and compiled, may not be able to instrument libraries
  - Can be done: manually, by the compiler, or automatically by a parser

Sources of Performance Information

- Sampling:
  - Requires virtually no changes made to program
  - Execution of the program is halted periodically via an interrupt source (usually a timer) and location of the program counter is recorded and optionally call stack is unwound
  - Allows attribution of time (or other interrupt source) to routines and lines of code based on the number of time the program counter at interrupt observed in routine or at line
  - Estimation of time attribution is not exact but with enough samples error is negligible
  - Require debugging information in executable to attribute below routine level
  - Performance data can be collected for entire program including libraries

Sources of Performance Information

- Library interposition:
  - A call made to a function is intercepted by a wrapping performance tool routine
  - Allows information about the intercepted call to captured and recorded, including: timing, call counts, and argument information
  - Can be done for any routine by one of several methods:
    - Some libraries (like MPI) are designed to allow calls to be intercepted. This is done using weak symbols (MPI_Send) and alternate routine names (PMPI_Send)
    - LD_PRELOAD – can force shared libraries to be loaded from alternate locations containing interposing routines, which then call actual routine
    - Linker Wrapping – instruct the linker to resolve all calls to a routine (ex: malloc) to an alternate wrapping routine (ex: malloc_wrap) which can then call the original routine

Sources of Performance Information

- Hardware Performance Counters:
  - Hardware register implemented on the chip that record counts of performance related events:
    - FLOPS - Floating point operations
    - L1 cache miss – number of L1 requests that miss
  - Processors support a very limited set of counters and countable events are preset by chip designer
  - Counters capabilities and events differ significantly between processors
  - Need to use an API to configure, start, stop, and read counters
  - Blue Gene/P:
    - Has 256 performance counter registers
    - Counters can only count for half the cores on a node at a time
    - BG/P events: BGP_PU0_FPU_MULT_1, BGP_PU1_DATA_LOADS
      - Complete list at wiki.alcf.anl.gov/index.php/UPC_Events_All

gprof

- Widely available Unix tool for collecting timing and call graph information via sampling
- Collects information on:
  - Approximate time spent in each routine
  - Count of the number times a routine was invoked
  - Call graph information:
    - list of the parent routines that invoke a given routine
    - list of the child routines a given routine invokes
    - estimate of the cumulative time spent in the child routines
- Advantages: widely available, easy to use, robust
- Disadvantages: scalability, opens one file per rank, doesn’t work with threads

gprof

- Using gprof:
  - Compile all and link all routines with the ‘-pg’ flag

    ```
    mpixlc -pg -O2 test.c ….
    mpixlf90 -pg -O2 test.f90 …
    ```

  - Run the code as usual: will generate one gmon.out for each rank
  - View data in gmon.out files, by running:

    ```
    gprof <executable-file> gmon.out.<id>
    ```

    - flags:
    - -p : display flat profile
    - -q : display call graph information
    - -l : displays instruction profile at source line level instead of function level
    - -C : display routine names and the execution counts obtained by invocation profiling
- Documentation at: wiki.alcf.anl.gov/index.php/Profiling

TAU

- The TAU (Tuning and Analysis Utilities) Performance System is a portable profiling and tracing toolkit for performance analysis of parallel programs
- TAU gathers performance information while a program executes through instrumentation of functions, methods, basic blocks, and statements via:
  - automatic instrumentation of the code at the source level using the Program Database Toolkit (PDT)
  - automatic instrumentation of the code using the compiler
  - manual instrumentation using the instrumentation API
  - at runtime using library call interception

TAU

- Some of the types of information that TAU can collect:
  - time spent in each routine
  - the number of times a routine was invoked
  - counts from hardware performance counters via PAPI
  - MPI profile information
  - Pthread, OpenMP information
  - memory usage

TAU

- Using TAU:
  - Choose how you want to instrument and what information you want to collect by setting the environment variables:

    TAU_MAKEFILE = /soft/apps/tau/tau-$TAUVERSION/bgp/lib/Makefile.tau-bgptimers-mpi-pdt

    TAU_OPTIONS = '-optVerbose -optNoRevert’
  - Compile with the TAU compiler wrappers:

    ```
    mpif90/mpixlf90 -> tau_f90.sh
    mpif77/mpixlf77 -> tau_f90.sh (add -qfixed for Fortran77)
    mpicc/mpixlc -> tau_cc.sh
    mpicxx/mpixlcxx -> tau_cxx.sh
    ```
  - Run the code:
    - Some environment variables may be set to control runtime behavior

      ```
      TAU_COMM_MATRIX= {0, 1}
      ```
    - Will generate files containing

TAU

- Using TAU (continued)
  - View collected performance data using TAU viewers:
    - paraprof: GUI environment for viewing collected performance information
    - pprof: command line tool providing summary information
    - PerfExplorer: framework for data mining and knowledge discovery
    - Viewers can be run on BG/P (with X-11 forwarding) or installed on local machine for better performance
- Documentation: wiki.alcf.anl.gov/index.php/Tuning_and_Analysis_Utilities_(TAU)

Rice HPCToolkit

- A performance toolkit that utilizes statistical sampling for measurement and analysis of program performance
- Assembles performance measurements into a call path profile that associates the costs of each function call with its full calling context
- Samples timers and hardware performance counters
  - Time bases profiling is well supported on the Blue Gene/P
  - Hardware performance counter profiling is limited to core 0 due to limitations of BG/P hardware
- Traces can be generated from a time history of samples
- Viewer provides multiple views of performance data including: top-down, bottom-up, and flat views

Rice HPCToolkit

- Using Rice HPCToolkit
  - Compile object files as usual adding '-g' flag to include debug info:
    ```
    mpixlc -g -O4 -qnoipa -c routine1.c
    ```
  - Link as usual but preface with the *hpclink* command:
    ```
    hpclink mpixlc -o myprog routine1.o ... -l<lib> ...
    ```
  - Run program specifying data to collect, produces *hpctoolkit-myprogram-measurements* directory containing performance data:
    ```
    qsub … --env HPCRUN_EVENT_LIST="WALLCLOCK@5000" myprog
    ```
  - Gather information from the executable with *hpcstruct,* produces *.hpcstruct* file:
    ```
    hpcstruct myprogram
    ```
  - Correlate collected performance data with information from executable, will produce *hpctoolkit-myprogram-database*:
    ```
    hpcprof -S myprogram.hpcstruct -I "path-to-myprogram-src/*"
      hpctoolkit-myprogram-measurements-XXXXXX
    ```

Rice HPCToolkit

- Using Rice HPCToolkit (continued)
  - View performance data:
    ```
    hpcviewer hpctoolkit-myprogram-database
    ```
  - Viewer can be run on BG/P (with X-11 forwarding) or installed on local machine for better performance

- Documentation: wiki.alcf.anl.gov/index.php/Rice_HPCToolkit

IBM HPCT

- A package from IBM that includes several performance tools:
  - MPI Profile and Trace library
  - Xprofiler
  - Hardware Performance Monitor (HPM) API
  - POMP OpenMP Profiling

- Documentation:
  - wiki.alcf.anl.gov/index.php/Performance_Tools
  - Redbook: High Performance Computing Toolkit for Blue Gene/P

IBM HPCT

- MPI Profiling and Tracing library
  - Collects profile and trace information about the use of MPI routines during a programs execution via library interposition
  - Profile information collected:
    - Number of times an MPI routine was called
    - Time spent in each MPI routine
    - Message size histogram
  - Tracing can be enabled and provides a detailed time history of MPI events
  - Point-to-Point communication pattern information can be collected in the form of a communication matrix

IBM HPCT

- Using the MPI Profiling and Tracing library
  - Recompile the code to include the profiling and tracing library:

```
mpixlc -o test-c test.c -g -L/soft/apps/current/ibm-hpct/lib -
lmpitrace -llicense -lgetarg

mpixlf90 -o test-f90 test.f90 -g -L/soft/apps/current/ibm-hpct/lib -
lmpitrace -llicense
```

  - Run the code as usual, optionally can control behavior using environment variables.
    - OUTPUT_ALL_RANKS={yes,no}
    - TRACE_ALL_TASKS={yes,no}
    - TRACE_ALL_EVENTS={yes,no}
  - Default output is MPI information for 4 ranks – rank 0, rank with max MPI time, rank with MIN MPI time, rank with MEAN MPI time. Can get output from all ranks by setting environment variables.

mpiP

- mpiP is a lightweight profiling library for MPI applications
- mpiP provides MPI information broken down by program call site:
  - the time spent in various MPI routines
  - the number of time an MPI routine was called
  - information on message sizes

- Documentation: wiki.alcf.anl.gov/index.php/Mpip

mpiP

- Using mpiP:
  - Recompile the code to include the mpiP library

```
mpixlc -o test-c test.c -L/soft/apps/current/mpiP/lib/ -lmpiP -L /bgsys/
drivers/ppcfloor/gnu-linux/lib -lbfd -liberty -unwind

mpixlf90 -o test-f90 test.f90 -L/soft/apps/current/mpiP/lib/ -lmpiP -L/bgsys/
drivers/ppcfloor/gnu-linux/lib -lbfd -liberty -unwind
```

  - Run the code as usual, optionally can control behavior using environment variable MPIP
  - Output is a single .mpiP summary file containing MPI information for all ranks

PAPI

- The Performance API (PAPI) specifies a standard API for accessing hardware performance counters available on most modern microprocessors
- PAPI provides two interfaces to counter hardware:
  - simple high level interface
  - more complex low level interface providing more functionality
- Defines two classes of hardware events:
  - PAPI Preset Events – Standard predefined set of events that are typically found on many CPU's. Derived from one or more native events. (ex: PAP_FP_OPS – count of floating point operations)
  - Native Events – Allows access to all platform hardware counters (ex: PNE_BGP_PU0_FPU_ADD_SUB_1 – Core zero standard PowerPC Add/Subtract instructions)

PAPI

- The BG/P PAPI implementation is not fully consistent with the PAPI standard semantics due to limitation of the BG/P counter hardware
  - Counts only valid for SMP mode with zero or 1 thread
  - Event counts are not reported in process/core context
  - Incorrect/incomplete definition of PAPI preset events
- PAPI implementation does have definitions for useful events not defined anywhere else:
  - Cache events such as:
    - PAPI_L1_DCH - Level 1 data cache hits
    - PAPI_L2_DCM - Level 2 data cache misses

PAPI

- To Use:
  - Determine which events to count:
    - Can run papi_avail utility *on compute nodes* to get a list of available PAPI preset and native events
  - Instrument code with calls to PAPI API
  - Link code with PAPI library (/soft/apps/current/papi /lib/libpapi.a)
  - Run the code

- Documentation:

  wiki.alcf.anl.gov/index.php/Performance_Application_Programming_Interface_(PAPI)

HPM

- Hardware Performance Monitor (HPM) is an IBM utility and API for accessing hardware performance counters
  - Configures, controls, and reads hardware performance counters
  - On Blue Gene/P only the API is available (no hpmcount utility)
- API provides a small number of calls to initialize, start, stop, and output information from the performance counters

```
void HPM_Init(void) - initialize the UPC unit
void HPM_Start(char *label) - start counting in a block marked by label
void HPM_Stop(char *label) - stop counting in a block marked by label
void HPM_Print(void) - print performance counter data
```

- Two slightly different version of the API are available on BG/P
  - libhpm.a – requires all 4 calls be used, creates 1 output file per rank
  - Libmpihpm.a – requires only the start/stop calls, creates aggregated output file with derived FLOP and memory metrics, also collects MPI information

HPM

- To Use:
  - Insert call to HPM API in code sections of interest
  - Link with HPM library:

```
mpixlc -o test-c test.c -L/soft/apps/current/hpm/lib/ -lhpm
mpixlc -o test-c test.c -L/soft/apps/current/hpm/lib/ -lmpihpm

mpixlf90 -o test-f test.f90 -L/soft/apps/current/hpm/lib/ -lhpm
mpixlf90 -o test-f test.f90 -L/soft/apps/current/hpm/lib/ -lmpihpm
```

  - Run code as usual, files output at program termination are:
    - libhpm – one hpm output file per rank or process
    - Libmpihpm – aggregated hpm file containing counter data, small number of files containing MPI information for selected ranks
- Documentation: wiki.alcf.anl.gov/index.php/High_Level_UPC_API

Darshan

- Darshan is a library that collects information on a programs IO operations and performance
- All applications compiled using the default MPI wrappers will use Darshan at run time without any manual steps from the user
- Upon successful job completion Darshan data is written to a file named

  <USERNAME>_<BINARY_NAME>_<COBALT_JOB_ID>_<DATE>.darshan.gz

  in the directory:
  - Surveyor: */pvfs-surveyor/logs/darshan/<YEAR>/<MONTH>/<DAY>*
  - Intrepid: */intrepid-fs0/logs/darshan/<YEAR>/<MONTH>/<DAY>*

Darshan

- Darshan data can be viewed by running the summary script:

```
darshan-job-summary.pl /pvfs-surveyor/logs/darshan/2009/7/27/
    carns_my-app_id114525_7-27-58921_19.darshan.gz --output ~/
    job-summary.pdf
```

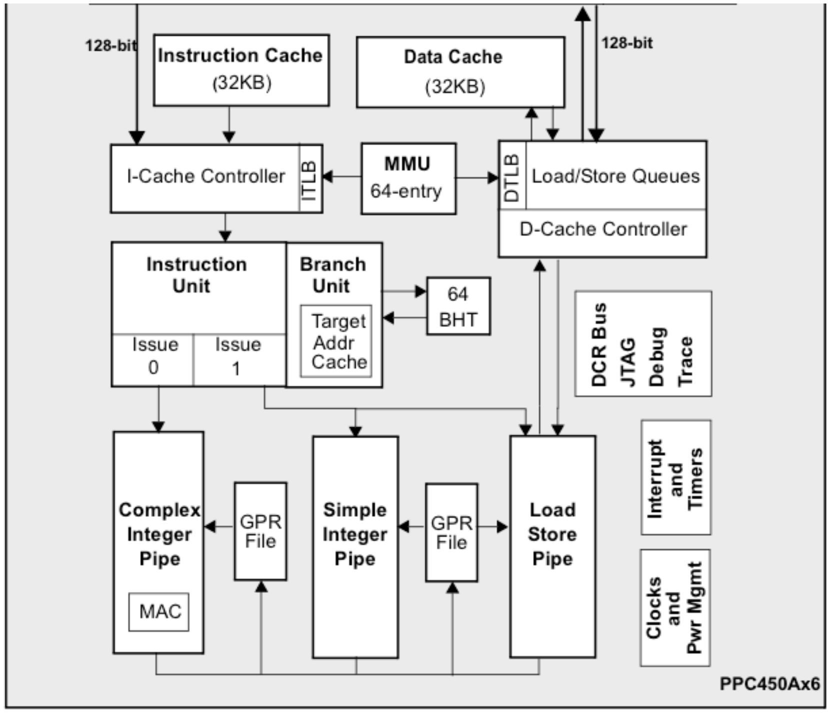
- Documentation:

  wiki.mcs.anl.gov/Darshan/index.php/Documentation_for_ALCF_users

Steps in looking in looking at performance

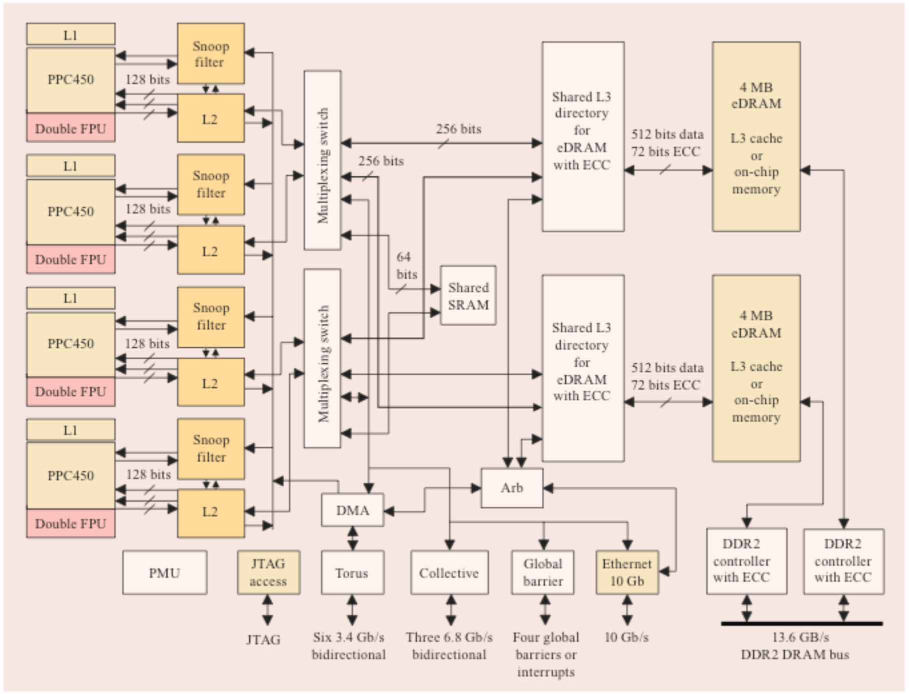
- Time based profile with at least one of the following:
  - Rice HPCToolkit
  - TAU
  - Gprof
- MPI profile with:
  - IBM HPCT MPI Profiling Library
  - mpiP
  - TAU
- Gather performance counter information for critical routines:
  - PAPI
  - HPM

BG/P CPU



- 7 Stage instruction pipeline:
  - Instruction Fetch
  - Instruction Decode
  - Issue
  - Register Access
  - Pipeline line stage 1
  - Pipeline line stage 2
  - Write Back
- Pipelines:
  - Load/Store (L-Pipe)
  - Simple Integer (J-Pipe)
  - Complex Integer (I-Pipe)
  - Floating Point
    - FMA
    - Double Hummer
- Dual Issue – can issues two instructions per cycle, must be to different pipelines

BG/P Memory Hierarchy



- Cost of memory access is now dominant factor in performance
- Blue Gene/P memory hierarchy:
  - L1 Instruction and L1 Data caches:
    - 32 KB total size, 4 cycle latency, 32-Byte line size, 64-way associative, round-robin
  - L2 Data cache:
    - 2KB prefetch buffer, 12 cycle latency, 16 lines, 128-byte line size
  - L3 Data cache:
    - 8 MB, 50 cycles latency, 128-byte line size, 8 way associative
  - Memory:
    - 2GB DDR-2 at 425 MHz, 104 cycles

Improving Performance

- Key to achieving high performance is optimal architecture-algorithm mapping
- Cost of memory access dominates performance – memory is slow compared to register operations
  - Pay close attention to data layout to maximize data locality
- Maximize instruction issue rate
- Utilize multi-flop floating point instructions:
  - FMA – fused multiply-add
  - Double Hummer

Improving Performance
Compiler

- Try several different compiler optimization levels to get best performance
- Compiler pragmas and functions:
  - #pragma disjoint(*a, *b) – specify that pointers a and b are disjoint
  - __alignx(), ALIGNX() – specify the alignment of variables
- Built-in and intrinsic procedures:
  - C: __lpfd(),,__fpmul(), __fpmadd() …
  - Fortran: LOADPF(), FPMUL(), FPMADD() …
- Inline assembly with asm keyword
- Look at what the compiler is doing
  - -qreport – produces a list file showing how code was optimized
  - -qlist – produces a list file showing object code

Improving Performance
Compiler Optimizations

- Common sub-expression elimination – removes redundant calculations of sub-expressions
- Strength reduction – replaces expensive operations with less expensive operations
- Loop invariant code motion – moves calculations outside of loops if not dependent on loop iteration
- Constant folding – computes the values of constant expressions at compile time
- Constant value propagation – replaces variables with constants when having constant value
- Induction variable simplification – simplifies computation iteration dependent variables in loops
- Loop unrolling – performs multiple loop steps in a single iteration, enables software pipelining and lessens loop overhead
- Dead code elimination – removes unused code
- Register allocation and instruction scheduling – optimizes ordering of instructions and register assignment

Improving Performance Libraries

- Use optimized libraries where ever possible:
  - MASS, MASSV
  - ESSL
  - BLAS/LAPACK
  - ScaLAPACK
  - fftw
  - p3dfft

Improving Performance
Floating Point Optimization Techniques

- Loop unrolling
- Software pipelining
- Improve ration of floating point to load/store operations
- Reduce register spilling
- Aliasing
- Fortran array syntax (operations, copy overhead)
- Eliminate floating point exceptions
- Eliminate unnecessary type conversions
- Avoid expensive special functions (sin, log, sqrt, …)
- If statements in loops
- Inline subroutine calls

Improving Performance
Memory Access Optimization Techniques

- General guideline - *Try to increase the spatial and temporal locality of references*
- Data layout
- Loop re-ordering
- Loop fusion
- Minimize stride
- Blocking
- Watch for cache trashing
- Pre-fetching

Improving Performance
MPI

- Load balance
- Remove unnecessary barriers
- MPI Datatypes for non-continuous data
- Avoid buffer copies
- Aggregate small messages
- Minimize surface to volume ratio of decomposition
- Post receives early
- Use non-blocking send/receives and overlap communication with computation
- Mapping onto network topology